Lösningsförslag till tentamen *P r e l i m i n ä r*

Kurs	**Objektorienterad programutveckling, fk**
Tentamensdatum	**2018-04-04**
Program	**DAI2**
Läsår	**2017/2018, lp 2**
Examinator	**Uno Holmer**

Uppgift 1 (7 p)

Metoden i tesen bryter mot designprincipen *separation of concern*. Den löser fyra olika uppgifter: Beräknar medelvärdet av fältelementen, väljer väg beroende på vad medelvärdet blev, samt utför två olika beräkningar (normaliseringar) av fälten. Detta kan delas upp i metoderna:

```
public void funcRefactored(int[] arr) {
    normalize(arr,computeMean(arr));
}

private double computeMean(int[] arr) {
    int sum = 0;
    for ( int x : arr )
        sum += x;
    return sum/(double)arr.length;
}

private void normalize(int[] arr,double mean) {
    if ( mean < 10 )
        normalizeLow(arr,mean);
    else if ( mean > 20 )
        normalizeHigh(arr,mean);
}

private void normalizeLow(int[] arr,double mean) {
    for ( int i = 0; i < arr.length; i++)
        arr[i] = arr[i] + 42;
}

private void normalizeHigh(int[] arr,double mean) {
    for ( int i = 0; i < arr.length - 1; i++)
        arr[i] = (arr[i] + arr[i+1])*2;
}
```

Uppgift 2 (6 p)

```
public class MySingletonClass {
    private Set<String> set;
    private static MySingletonClass instance = null;
    private MySingletonClass() {
        set = new HashSet<>();
    }
    public synchronized static MySingletonClass getInstance() {
        if ( instance == null )
            instance = new MySingletonClass();
        return instance;
    }
    public void add(String s) {
        set.add(s);
    }
    public boolean contains(String s) {
```

```
        return set.contains(s);
    }
}
```

Uppgift 3 (8 p)

Designen bryter mot designprinciperna *DIP* och *LSP*. Fallanalysen över olika personalkategorier är oflexibel och typosäker. Om metoderna take och receive unifieras kan vi införa ett övergripande gränssnitt för anställda. Även bonusklasserna bör ha ett övergripande gränssnitt.

```
public interface Employee {
    void receive(Bonus x);
    public int yearsEmployed();
}
public class Academic implements Employee {
    public void receive(Bonus x) { ... }
    public int yearsEmployed() { ... }
}
public class OfficeClerk implements Employee {
    public void receive(Bonus x) { ... }
    public int yearsEmployed() { ... }
}

public interface Bonus {}
public class CheapBonus implements Bonus { ... }
public class DefaultBonus implements Bonus { ... }
public class MediumBonus implements Bonus { ... }
public class ExpensiveBonus implements Bonus { ... }
```

Koden som skapar bonusobjekten med utgångspunkt i anställningsår är starkt beroende av specifika typnamn. Detta placeras lämpligen i en objektfabrik.

```
public class BonusFactory {
    public static Bonus createBonus(int years) {
        if ( years > 30 )
            return new ExpensiveBonus();
        else if ( years > 20 )
            return new MediumBonus();
        else if ( years > 10 )
            return new CheapBonus();
        else
            return new DefaultBonus();
    }
}
```

Metoden `giveBonus` kan nu skrivas om på ett typsäkert sätt genom att tillämpa polymorfism:

```
public static void giveBonus(ArrayList<Employee> l) {
    for ( Employee e : l )
        e.receive(BonusFactory.createBonus(e.yearsEmployed()));
}
```

Uppgift 4 (8 p)

```
public abstract class AbstractDieDecorator implements Die {
    private Die die;
    public AbstractDieDecorator(Die die) {
        this.die = die;
    }
    @Override
    public int getValue() {
        return die.getValue();
    }
    @Override
    public void roll() {
        die.roll();
    }
}

public class HistoryDie extends AbstractDieDecorator {
    private LinkedList<Integer> history;
    private final int capacity;

    public HistoryDie(Die die,int capacity) {
        super(die);
        this.capacity = capacity;
        history = new LinkedList<>();
    }
    @Override
    public void roll() {
        super.roll();
        if ( capacity > 0 ) {
            if ( history.size() == capacity )
                history.removeLast();
            history.addFirst(getValue());
        }
    }
    public int available() {
        return history.size();
    }
    public int lookBack(int n) throws IndexOutOfBoundsException {
        if ( n < 0 || n >= available() )
            throw new IndexOutOfBoundsException("HistoryDie: index out of
range");
        return history.get(n);
    }
}
```

Uppgift 5 (8 p)

I gruppklassens `add`-metod adderas *muterbara* rektangelobjekt till gruppens interna lista. Inget hindrar att sådana objekt delas av flera gruppobjekt. När `resize` anropas för ett gruppobjekt kan därför ett annat muteras implicit genom att ett delat rektangelobjekt muteras av `resize`. Gruppklassens klassinvariant håller alltså inte.

```
Group g = new Group();
Rectangle r = new Rectangle(10,20);
g.add(r);
System.out.println("Total area of g: " + g.getArea());
Group g2 = new Group();
g2.add(r);  // r delas nu av både g och g2
g2.resize(2); // Invarianten bryts
System.out.println("Total area of g: " + g.getArea());
```

En lösning på problemet är att göra rektangelklassen kopierbar och låta gruppklassen addera kopior av rektangelobjekt.

```
public class Rectangle implements Cloneable {
    ...
    public Rectangle clone() {
        try {
            return (Rectangle)super.clone();
        }
        catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
}
public class Group {
    ...
    public void add(Rectangle r) {
        checkInvariant();
        allRectangles.add(r.clone());
        totalArea += r.getArea();
        checkInvariant();
    }
    ...
}
```

Uppgift 6 (3+4 p)

a)
Metoden `client` har en parameter av typen `Int` och kan anropas med vilket implementerande objekt som helst förutsatt att dess `service`-metod har minst lika stark specifikation som i `Int`. Den enda klassen som uppfyller detta är `Impl4`. I `Impl1` är specifikationen svagare och i de två övriga ojämförbara.

b)
I en klass som implementerar `Int` måste `func` i subklassen:

- vara minst lika synlig som i `Int`
- ha en returtyp som är en subtyp till `B` (kovarians)
- ha samma parametertyp
- kasta kompatibla undantag: inga undantag, `E2`, `E3` eller `E2` och `E3`

Detta uppfylls av fallen a,b,e, och f.

Uppgift 7 (8 p)

Variabeln `p` har *statisk* typ `A`. Den *dynamiska* typen går från `A` till `B` och slutligen till `C`. Variabelns statiska typ bestämmer vilken *klassmetod* som anropas, dess dynamiska typ vilken *instansmetod* som anropas. Metoden `h` *överlagras* i `A`. Vid överlagring matchas anropet mot den metod vars parameterprofil överensstämmer bäst med typerna i anropet. Därefter anropas den valda metoden eller en överskuggning av den. Utskrifterna blir:

```
A p = new A();                // Utskrift
p.f();                        // A.f      trivialt
p.g();                        // A.g      trivialt
p = new B();
p.f();                        // B.f      f är en instansmetod omdef. i B
p.g();                        // A.g      g är en klassmetod
p.h("apa");                   // A.h(String) apa      1)
p.h(123);                     // B.h(Object) 123      2)
p.i("bepa");                  // A.i(Object)bepa      3)
p = new C(42);
print(p.equals(new C(42)));   // false                4)
```

1) `h` överlagras i `A`. Anropet matchar `A.h(String)` vilken anropas.
2) Anropet matchar `A.h(Object)` som överskuggas av `B.h(Object)` vilken anropas.
3) Anropet matchar `A.i(Object)` vilken anropas eftersom den inte överskuggas i B.
4) Anropet matchar `Object.equals(Object)` vilken anropas. Den överskuggas inte i `C`.
   Anropet `p.equals(new C(42))` returnerar förstås `false` eftersom `Object.equals` baseras på
   referenslikhet.

Uppgift 8 (8 p)

```
public static void main(String[] arg) {
    try {
        copyBinaryToText(arg[0]);
    }
    catch ( FileNotFoundException e ) {
        System.out.println("Cannot open " + e.getMessage());
    }
    catch ( IOException e ) {
        e.printStackTrace();
    }
}

private static void copyBinaryToText(String inFileName)
throws IOException
{
    DataInputStream in = openBinaryInFile(inFileName);
    PrintWriter out = createTextOutFile(inFileName + ".txt");
    copyFile(in,out);
    in.close();
    out.close();
}

private static DataInputStream openBinaryInFile(String fileName)
throws IOException
{
    return new DataInputStream(new FileInputStream(fileName));
}

private static PrintWriter createTextOutFile(String fileName)
```

```
throws IOException
{
    return new PrintWriter(new FileWriter(fileName));
}

private static void copyFile(DataInputStream in,PrintWriter out)
throws IOException
{
    while ( in.available() > 0 ) {
        long value = in.readLong();
        out.println("" + value);
    }
}
```